CS388G Problem Set #3

Nidhi Kadkol (nk9368)

March 2019

1(a)

Proof. Suppose we make an access to key x_k. Then x_k is at the root of the tree and is moved to the front of the list. The size of the root is the sum of the weights of all the nodes in the tree which is $\sum_{i=1}^{n} 1/i^2$. This is the same before and after the weight reassignment procedure. Hence, $s^*(x_k) = s(x_k)$ where s^* represents the size of a node after weight reassignment. If x_k was accessed in the previous access as well, then it is already at the root and there is no reordering in the tree or the MTF list. So the weights (and hence the sizes) of all the nodes remain the same. Else x_k now moves to the front of the list. This causes all the other nodes to increase their position by 1, thus for a node in position i, its weight decreases from $1/i^2$ to $1/(i+1)^2$. So, for all the non-root nodes, their weight decreases. This means that the sizes of all the non-root nodes decrease as well. Hence,

$$s^*(x_k) = s(x_k) \qquad \text{where } x_k \text{ is the root node} \tag{1}$$

$$s^*(x_j) \leq s(x_j) \qquad \forall\, j \neq k \tag{2}$$

From equations (1) and (2) we get

$$\begin{aligned}
& s^*(x_i) \leq s(x_i) && \forall\, i,\ 1 \leq i \leq n \\
\implies & \log(s^*(x_i)) \leq \log(s(x_i)) && \forall\, i,\ 1 \leq i \leq n \\
\implies & r^*(x_i) \leq r(x_i) && \forall\, i,\ 1 \leq i \leq n \\
\implies & \sum_{i=1}^{n} r^*(x_i) \leq \sum_{i=1}^{n} r(x_i) && \\
\implies & \Phi^*(T) \leq \Phi(T) &&
\end{aligned}$$

Thus, the weight reassignment procedure performed after an access does not increase the potential. ■

1(b)

Proof. The minimum size of a node is when it is a leaf and its position in the MTF list is n. Its size is $1/n^2$. The maximum possible size of a node is when it is the root. Its size is $\sum_{i=1}^{n} 1/i^2$. Thus, the minimum and maximum rank of a node are $\log(1/n^2)$ and $\log(\sum_{i=1}^{n} 1/i^2)$ respectively. We know that

$$\sum_{i=1}^{n} 1/i^2 = (1 + 1/2^2 + 1/3^2 + \cdots 1/n^2) \leq 2$$

The maximum drop in rank of a node is thus

$$\begin{aligned}
&\log\left(\sum_{i=1}^{n} 1/i^2\right) - \log(1/n^2) \\
=&\log\left(\sum_{i=1}^{n} 1/i^2\right) + \log(n^2) \\
=&\log\left(\sum_{i=1}^{n} 1/i^2\right) + 2\log(n) \\
\leq&\log 2 + 2\log(n)
\end{aligned}$$

The maximum drop in potential is equal to the sum of the maximum drop in potential for each of the nodes. This is bounded by

$$\begin{aligned}
&n \cdot (\log 2 + 2\log(n)) \\
=&n\log 2 + 2n\log(n) \\
=&O(n\log n)
\end{aligned}$$

■

1(c)

Proof. Consider access i. Suppose it has accessed key x. There are 2 cases - either x has been accessed before or x is being accessed for the first time. If x was accessed before, then in its previous access it was moved to the front of the list. At access i, there have been t_i accesses to distinct keys that are not x. Each of these accesses pushed x by one position down the list, so by access i, x is at position $1 + t_i$. Thus, $w(x) = 1/(1 + t_i)^2$. The size of x

is the sum of the weights of all the nodes in its subtree. Hence,

$$\begin{aligned}
& s(x) \geq 1/(1+t_i)^2 \\
\implies \quad & r(x) \geq \log\left(\frac{1}{(1+t_i)^2}\right) \\
\implies \quad & r(x) \geq -2\log(1+t_i) \\
& -r(x) \leq 2\log(1+t_i) \\
& r(t) - r(x) \leq r(t) + 2\log(1+t_i) \\
\implies \quad & r(t) - r(x) \leq \log 2 + 2\log(1+t_i) \qquad \text{since } r(t) \leq \log 2 \\
\implies & 3(r(t) - r(x)) + 1 \leq 3\log 2 + 6\log(1+t_i) + 1
\end{aligned}$$

This is the amortized cost for a single access i. For all m access, we get the sum of amortized costs is

$$\begin{aligned}
& (3\log 2)m + 6\sum_{i=1}^{m} \log(1+t_i) + m \\
=\ & (3\log 2 + 1)m + 6\sum_{i=1}^{m} \log(1+t_i) \\
=\ & O(m + \sum_{i=1}^{m} \log(1+t_i))
\end{aligned}$$

Adding the drop in potential, we get that the total cost is

$$O(m + n\log n + \sum_{i=1}^{m} \log(1+t_i))$$

In the case where x_i was not accessed before access i we have

$$\begin{aligned}
& w(x) \geq 1/n^2 \\
\implies \quad & s(x) \geq 1/n^2 \\
\implies \quad & r(x) \geq \log(1/n^2) = -2\log n \\
& -r(x) \leq 2\log n \\
& r(t) - r(x) \leq r(t) + 2\log n \\
& \leq \log 2 + 2\log n \\
\implies & 3(r(t) - r(x)) + 1 \leq 3\log 2 + 6\log n + 1
\end{aligned}$$

For m accesses, we get

$$\begin{aligned}
& (3\log 2)m + 6m\log n + m \\
=\ & (3\log 2 + 1)m + 6m\log n
\end{aligned}$$

In this case $m \leq n$, as we cannot have more than n accesses and still have x_i being accessed for the first time. Hence, the sum of the amortized costs is bounded by

$$\begin{aligned} &(3\log 2 + 1)n + 6n\log n \\ =\, &O(n\log n) \end{aligned}$$

This is a subset of the terms in the O-bound of the first case. So combining both the cases together, we get that the total cost is

$$O(m + n\log n + \sum_{i=1}^{m} \log(1 + t_i))$$

■

2

We choose a Red-Black Tree as our underlying data structure in which the primary key of a node is x and the secondary key is y. The number of nodes in the tree will be equal to the number of integer pairs in the set $= |S|$. So inserting and deleting an element takes $O(\log |S|)$ time, done using the usual Red-Black tree method of inserting and deleting nodes ordered according to their keys in the tree. For finding an integer pair we first search for the x value in the tree. Once we do that, we check if the y value of the node is equal to the value we are searching for. If not, we continue to search from that node for a node with the same x and y value. If we encounter a node where the x is no longer the same, it means the pair we are searching for is not present in the tree and we return Like a usual red-black tree find, this operation takes $O(\log |S|)$ time.

To take care of the sum operation, we augment the tree by storing 2 additional pieces of information at each node p, that is $p.sum\_left$ and $p.sum\_right$ which store the sum of y values of all the nodes in the left and right subtrees of p respectively.

We first show that we can maintain the information for the original operations in the same asymptotic time. A find operation does not depend on these extra attributes so its complexity does not change. If p is a leaf node, then its $left\_sum$ and $right\_sum$ are 0. Otherwise, note that for a node p in the tree, we calculate the extra attributes as

$$\boldsymbol{p.sum\_left = p.left.sum\_left + p.left.sum\_right + p.left.x}$$

$$\boldsymbol{p.sum\_right = p.right.sum\_left + p.right.sum\_right + p.right.x}$$

We see that the auxiliary attributes of a node only depend on the information in its left child and right child. Thus, when we insert a new node with keys x and y into the tree, We can compute its $left\_sum$ and $right\_sum$ in $O(1)$ time. This will require us to update the $left\_sum$ and $right\_sum$ attributes for its parent, and this will propagate upto the root of the

tree. Since the height of a red-black tree is $O(\log n)$ where n is the number of nodes, therefore this phase of insertion takes $O(\log|S|)$ time. After this, we perform at most 2 rotations to complete the insertion step. Since a rotation changes only 2 nodes, we take $O(\log|S|)$ to propagate the changes to $left\_sum$ and $right\_sum$ upto the root for each rotation, and thus take $O(\log|S|)$ in total for an insertion step.

For deletion, we first remove the node to be deleted from the tree. If the deleted node had children, its children become the children of the deleted node's successor. Since we have removed a node, we propagate the changes to $left\_sum$ and $right\_sum$ from the local changes upto the root. This takes $O(\log|S|)$ time. In the second phase, at most 3 rotations occur, and since the time for a rotation to propagate the changes to the root is $O(\log|S|)$, therefore the time for this phase is $O(\log|S|)$ and the total time for deletion of a node is $O(\log|S|)$.

Now, we show how to carry out the SUM operation. Given (a, b) as input, we first want to find the first valid node, that is a node p such that $a \leq p.x \leq b$. If the root is a valid node, then we start the next part of our algorithm from there. If it is not, then either $x < a$ or $x > b$. If $x < a$, we want to move to a node with greater x value, so we descend to the right child. If $x > b$, then we want to move to a node with lesser x value, so we descend to the left child. We continue this procedure until we reach the first valid node. If we reach the end of the tree and no node is valid, then we return 0. We have traversed the depth of the tree and the time complexity of this is $O(\log|S|)$.

Now suppose we have reached the first valid node p. Initialize a variable $sum = 0$. We set a new pointer to p (as we want to come back to it later) and add $p.y$ to sum. Then we move to the left child of p. If this node is also valid, then we add this node's sum_right to sum. This is because all the nodes in the right subtree of $p.left$ are greater than or equal to $p.left.x$ and less than or equal to $p.x$, which make them all valid nodes. We then move to the left child of $p.left$ again and add its sum_right if it is valid and then descend left. That is, we descend left from p and add the sum_right values of the nodes we descend through until we hit an invalid node. Suppose the invalid node we encounter is r. Then r is invalid because $r.x < a$. So we descend to the right child of r, as to move to a valid node we need a node with greater x. We keep descending right until we reach a valid node. We then add sum_right of this node, and then descend left, continuing the process. In essence, we traverse the depth of the tree, adding sum_left and descending left for valid nodes, and not adding anything and descending right for invalid nodes. This traversal thus takes $O(\log|S|)$ time since the depth of the red-black tree is $O(\log|S|)$.

Now, we come back to p in $O(1)$ time using the pointer we had set to p. We now descend right. If this node is also valid, we add this node's sum_left to sum. This is because all the nodes in the left subtree of $p.right$ are greater than or equal to p and less than or equal to $p.right$, making them all valid nodes. We then descend right from $p.right$ and if it is valid, add its sum_left and proceed right again, repeating the process. If we encounter an invalid node r, it means that $r.x > b$. So we descend left from r and do the same thing, that is, if $r.left$ is valid add its sum_left to sum and descend right, and if it is invalid, descend right without adding anything to sum. In this way, we traverse the depth of the tree until we

reach a leaf node. The traversal thus takes $O(\log|S|)$ time since the depth of the red-black tree is $O(\log|S|)$. Thus, the total cost of a SUM operation is $O(\log|S|)$.

3(a)

Let x be any node in a Fibonacci heap. Suppose that $x.degree = k$. Let $y_1, y_2, \cdots, y_k$ denote the children of x in the order in which they were linked to x, from the earliest to the latest.

We now prove that $y_i.degree \geq 0$ for $i = 1, 2$ and $y_i.degree \geq i - 3$ for $i = 3, 4, \cdots, k$.

Proof. $y_1.degree \geq 0$ as this is true for all nodes. Note that when y_2 is linked to x, y_1 was already a child of x. So $x.degree = 1$ before y_2 was added to the child list of x. Since a node is linked to x only if the degree of the node is equal to the degree of x, therefore $y_2.degree = 1$. Since then, y_2 can lose its child so $y_2.degree \geq 0$.

For $i \geq 3$, note that when y_i is linked to x, the nodes $y_1, y_2, \cdots, y_{i-1}$ were already children of x and so $x.degree = i - 1$. Since y_i can be linked to x only if its degree is equal to x, therefore $y_i.degree = i - 1$. Since then, y_i can lose at most 2 children as if it loses 3 children then it gets cut from x by the cascading cut procedure. Thus, $y_i.degree \geq i - 3$. ■

Now, let a_k denote the minimum size of a subtree rooted at a node with degree k. We see that $a_0 = 1$, $a_1 = 2$, and $a_2 = 3$. For bounding a_k with $k \geq 3$, we add 1 for the root node, 1 for y_1 (as size of $y_1 \geq 1$), and 1 for y_2 (as size of $y_2 \geq 1$ since degree of $y_2 \geq 0$). Thus we get

$$\begin{aligned} a_k &= 3 + \sum_{i=3}^{k} (\text{minimum size of } y_i) \\ &= 3 + \sum_{i=3}^{k} a_{y_i.degree} \\ &= 3 + \sum_{i=3}^{k} a_{i-3} \end{aligned}$$

Now notice that

$$\begin{aligned} a_{k-1} &= 3 + \sum_{i=3}^{k-1} a_{i-3} \\ &= 3 + a_0 + a_1 + \cdots + a_{k-4} \\ a_k &= 3 + \sum_{i=3}^{k} a_{i-3} \\ &= 3 + a_0 + a_1 + \cdots + a_{k-4} + a_{k-3} \\ &= a_{k-1} + a_{k-3} \end{aligned}$$

Therefore, our recurrence is

$$a_k = \begin{cases} k+1 & \text{if } 0 \le k < 3, \\ a_{k-1} + a_{k-3} & \text{if } k \ge 3. \end{cases}$$

3(b)

We have the recurrence

$$a_k = \begin{cases} k+1 & \text{if } 0 \le k < 3, \\ a_{k-1} + a_{k-3} & \text{if } k \ge 3. \end{cases}$$

We have to show that there is a constant c such that $a_k \ge c^k$. We shall prove that this is true for $c = 5/4 = 1.25$. We prove by induction that $a_k \ge c^k$ for all $k \ge 0$.

Proof. **Base Case:** When $k = 0$, we have $a_0 = 1$ and $c^0 = 1$. Thus, $a_0 \ge c^0$. When $k = 1$, we have $a_1 = 2$ and $c^1 = 1.25$. Thus, $a_1 \ge c^1$. When $k = 2$, we have $a_2 = 3$ and $c^2 = 1.5625$. Thus, $a_2 \ge c^2$.

Induction Hypothesis: Suppose that $a_i \ge c^i$ for all $i = 0$ to $k-1$. Now we have to prove that $a_k \ge c^k$.

Induction Step:

$$\begin{aligned} a_k &= a_{k-1} + a_{k-3} \\ &\ge c^{k-1} + c^{k-3} \\ &= c^k(1/c + 1/c^3) \\ &= c^k(4/5 + 4^3/5^3) \\ &= c^k((4 \times 25 + 4^3)/5^3) \\ &= c^k((100 + 64)/125) \\ &= c^k(164/125) \\ &\ge c^k \end{aligned}$$

Hence, there is a constant $c = 1.25$ such that $a_k \ge c^k$. ■

4

When calculating shortest path distances from s to all other vertices v, we give each vertex v an attribute $v.d$ which is the weight of the shortest path estimate from s to v. Let $\delta(s, v)$ be the weight of the actual shortest path from s to v. Given 2 vertices u and v having $u.d$ and $v.d$, we relax the edge (u, v) to try and get a better shortest path estimate of v. In relaxing, we check if $u.d + w(u, v) < v.d$. If it is, we update $v.d$ to be equal to $u.d + w(u, v)$.

From the path relaxation property of weighted directed graphs we know that if we relax the edges from vertex s to vertex v in the order of their occurrence in the shortest path from s to v, then we get $v.d = \delta(s, v)$. That is, we get the correct value of the shortest path from s to v if the edges are relaxed in this order even if other edge relaxations occur before, after, or in between this sequence.

In the Bellman Ford algorithm, we choose an arbitrary sequence of edges and relax all of them in that order $|V| - 1$ times. This gives a running time of $O(VE)$.

We know that the weights of the edges along the shortest path from s to any v form a bitonic sequence. This means that the edge weights along any shortest path will do one of the following (where I is increase and D is decrease):

- Increase then decrease (ID)
- Decrease then increase (DI)
- Increase then decrease then increase again (IDI)
- Decrease then increase then decrease again (DID)

With this information, we can choose the order in which we want to relax the edges and not relax all the edges as many as $|V| - 1$ times. We sort all the edges. We first relax all the edges in decreasing order. Then we relax all the edges in increasing order. Then we relax all the edges in decreasing order again. Finally, we again relax all the edges in increasing order. Thus, these 4 iterations of going through the edges in the order $DIDI$ cover all the possible cases of the orderings of the shortest paths of the edges. Thus, at the end of these 4 relaxation sequences, we have $v.d = \delta(s, v)$ for all v.

Relaxing all the edges takes $O(E)$, and we do it 4 times. Sorting the edges in increasing or decreasing order takes $O(E \lg E)$ time, which we do 4 times at max. So the total run time would be $4\,O(E) + 4\,O(E \lg E) = O(E \lg E)$.